Overview and summary

We have discussed...

- What AI and intelligent agents are
- How to develop AI systems
- How to solve problems using search
- How to play games as an application/extension of search
- How to build basic agents that reason logically, using propositional logic
- How to write more powerful logic statements with first-order logic
- How to properly engineer a knowledge base
- How to reason logically using first-order logic inference
- Examples of logical reasoning systems, such as theorem provers
- How to plan
- Expert systems
- What challenges remain
Acting Humanly: The Turing Test

- **Alan Turing's** 1950 article *Computing Machinery and Intelligence* discussed conditions for considering a machine to be intelligent
  - “Can machines think?” $\leftrightarrow$ “Can machines behave intelligently?”
  - The Turing test (The Imitation Game): Operational definition of intelligence.

- **Computer needs to posses:** Natural language processing, Knowledge representation, Automated reasoning, and Machine learning
What would a computer need to pass the Turing test?

- **Natural language processing**: to communicate with examiner.
- **Knowledge representation**: to store and retrieve information provided before or during interrogation.
- **Automated reasoning**: to use the stored information to answer questions and to draw new conclusions.
- **Machine learning**: to adapt to new circumstances and to detect and extrapolate patterns.
- **Vision** (for Total Turing test): to recognize the examiner’s actions and various objects presented by the examiner.
- **Motor control** (total test): to act upon objects as requested.
- **Other senses** (total test): such as audition, smell, touch, etc.
What would a computer need to pass the Turing test?

- **Natural language processing**: to communicate with examiner.
- **Knowledge representation**: to store and retrieve information provided before or during interrogation.
- **Automated reasoning**: to use the stored information to answer questions and to draw new conclusions.
- **Machine learning**: to adapt to new circumstances and to detect and extrapolate patterns.
- **Vision (for Total Turing test)**: to recognize the examiner’s actions and various objects presented by the examiner.
- **Motor control (total test)**: to act upon objects as requested.
- **Other senses (total test)**: such as audition, smell, touch, etc.
What is an (Intelligent) Agent?

• Anything that can be *viewed as perceiving* its *environment* through *sensors* and *acting* upon that environment through its *effectors* to maximize progress towards its *goals*.

• PAGE (Percepts, Actions, Goals, Environment)

• Task-specific & specialized: well-defined *goals* and *environment*
## Environment types

<table>
<thead>
<tr>
<th>Environment</th>
<th>Accessible</th>
<th>Deterministic</th>
<th>Episodic</th>
<th>Static</th>
<th>Discrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Reality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Office Environment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mars</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Environment types

<table>
<thead>
<tr>
<th>Environment</th>
<th>Accessible</th>
<th>Deterministic</th>
<th>Episodic</th>
<th>Static</th>
<th>Discrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Virtual Reality</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes/No</td>
<td>No</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Office Environment</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Mars</td>
<td>No</td>
<td>Semi</td>
<td>No</td>
<td>Semi</td>
<td>No</td>
</tr>
</tbody>
</table>

The environment types largely determine the agent design.
Agent types

- Reflex agents
- Reflex agents with internal states
- Goal-based agents
- Utility-based agents
Reflex agents
Reflex agents w/ state
Goal-based agents
Utility-based agents

Agent

- State
- How the world evolves
- What my actions do
- Utility

Environment

- Sensors
- What the world is like now
- What it will be like if I do action A
- How happy I will be in such a state
- What action I should do now

Effectors
How can we design & implement agents?

- Need to study knowledge representation and reasoning algorithms
- Getting started with simple cases: search, game playing
Function `SIMPLE-PROBLEM-SOLVING-AGENT(p)` returns an action

**Inputs:** $p$, a percept

**Static:** $s$, an action sequence, initially empty

- $state$, some description of the current world state
- $g$, a goal, initially null
- $problem$, a problem formulation

$state \leftarrow UPDATE-STATE(state, p)$

**If** $s$ is empty **then**

- $g \leftarrow FORMULATE-GOAL(state)$
- $problem \leftarrow FORMULATE-PROBLEM(state, g)$
- $s \leftarrow SEARCH(problem)$
- $action \leftarrow RECOMMENDATION(s, state)$
- $s \leftarrow REMAINDER(s, state)$

**Return** $action$

Note: This is offline problem-solving. Online problem-solving involves acting w/o complete knowledge of the problem and environment.
Problem types

• **Single-state problem:** deterministic, accessible
  
  *Agent knows everything about world, thus can calculate optimal action sequence to reach goal state.*

• **Multiple-state problem:** deterministic, inaccessible
  
  *Agent must reason about sequences of actions and states assumed while working towards goal state.*

• **Contingency problem:** nondeterministic, inaccessible
  
  *Must use sensors during execution*
  
  *Solution is a tree or policy*
  
  *Often interleave search and execution*

• **Exploration problem:** unknown state space
  
  *Discover and learn about environment while taking actions.*
Search algorithms

Basic idea:

offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

**Function** General-Search(*problem, strategy*) returns a *solution*, or failure

initialize the search tree using the initial state problem

**loop do**

  *if* there are no candidates for expansion *then return* failure

  choose a leaf node for expansion according to strategy

  *if* the node contains a goal state *then return* the corresponding solution

  *else* expand the node and add resulting nodes to the search tree

**end**
Implementation of search algorithms

**Function** General-Search(problem, Queuing-Fn) returns a solution, or failure

nodes ← make-queue(make-node(initial-state[problem]))

loop do
    if node is empty then return failure
    node ← Remove-Front(nodes)
    if Goal-Test[problem] applied to State(node) succeeds then return node
    nodes ← Queuing-Fn(nodes, Expand(node, Operators[problem]))
end

**Queuing-Fn(queue, elements)** is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

**Solution:** is a sequence of operators that bring you from current state to the goal state.
Encapsulating *state* information in *nodes*

A *state* is a (representation of) a physical configuration. A *node* is a data structure constituting part of a search tree, includes *parent, children, depth, path cost* $g(x)$. *States* do not have parents, children, depth, or path cost!

The *Expand* function creates new nodes, filling in the various fields and using the *Operators* (or *SuccessorFn*) of the problem to create the corresponding states.
Complexity

• Why worry about complexity of algorithms?

➤ because a problem may be solvable in principle but may take too long to solve in practice

• How can we evaluate the complexity of algorithms?

➤ through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size $n$ of a problem when $n$ tends towards infinity
Why is exponential complexity “hard”?  

It means that the number of operations necessary to compute the exact solution of the problem grows exponentially with the size of the problem (here, the number of cities).

- \( \exp(1) = 2.72 \)
- \( \exp(10) = 2.20 \times 10^4 \) (daily salesman trip)
- \( \exp(100) = 2.69 \times 10^{43} \) (monthly salesman planning)
- \( \exp(500) = 1.40 \times 10^{217} \) (music band worldwide tour)
- \( \exp(250,000) = 10^{108,573} \) (fedex, postal services)
- Fastest computer \( = 10^{12} \) operations/second

In general, exponential-complexity problems cannot be solved for any but the smallest instances!
Landau symbols

f is dominated by g:

\[ f \in O(g) \iff \exists k, f(n) \leq k g(n) \iff \lim_{n \to \infty} \frac{f}{g} \text{ is bounded} \]

f is negligible compared to g:

\[ f \in o(g) \iff \forall k, f(n) \leq k g(n) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} \to 0 \]
Polynomial-time hierarchy

- From Handbook of Brain Theory & Neural Networks (Arbib, ed.; MIT Press 1995).

AC₀: can be solved using gates of constant depth

NC¹: can be solved in logarithmic depth using 2-input gates

NC: can be solved by small, fast parallel computer

P: can be solved in polynomial time

P-complete: hardest problems in P; if one of them can be proven to be
            NC, then P = NC

NP: non-polynomial algorithms

NP-complete: hardest NP problems; if one of them can be proven to be
            P, then NP = P

PH: polynomial-time hierarchy
Search strategies

**Uninformed:** Use only information available in the problem formulation
- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening

**Informed:** Use heuristics to guide the search
- Greedy search
- A* search

**Iterative Improvement:** Progressively improve single current state
- Hill climbing
- Simulated annealing
Search strategies

Uninformed: Use only information available in the problem formulation
- Breadth-first – expand shallowest node first; successors at end of queue
- Uniform-cost – expand least-cost node; order queue by path cost
- Depth-first – expand deepest node first; successors at front of queue
- Depth-limited – depth-first with limit on node depth
- Iterative deepening – iteratively increase depth limit in depth-limited search

Informed: Use heuristics to guide the search
- Greedy search – queue first nodes that maximize heuristic “desirability” based on estimated path cost from current node to goal
- A* search – queue first nodes that minimize sum of path cost so far and estimated path cost to goal

Iterative Improvement: Progressively improve single current state
- Hill climbing – select successor with highest “value”
- Simulated annealing – may accept successors with lower value, to escape local optima
Example: Traveling from Arad To Bucharest
Breadth-first search
Breadth-first search
Breadth-first search
Uniform-cost search
Uniform-cost search
Uniform-cost search
Depth-first search
Depth-first search
Depth-first search

I.e., depth-first search can perform infinite cyclic excursions.
Need a finite, non-cyclic search space (or repeated-state checking)
Iterative deepening search $l = 0$
Iterative deepening search $l = 1$
Iterative deepening search $l = 2$
Informed search: Best-first search

- **Idea:**
  
  use an evaluation function for each node; estimate of “desirability”

  \[ \Rightarrow \text{expand most desirable unexpanded node.} \]

- **Implementation:**

  \[ \text{QueueingFn} = \text{insert successors in decreasing order of desirability} \]

- **Special cases:**

  greedy search
  
  A* search
Greedy search

- **Estimation function:**
  
  \[ h(n) = \text{estimate of cost from } n \text{ to goal (heuristic)} \]

- **For example:**
  
  \[ h_{SLD}(n) = \text{straight-line distance from } n \text{ to Bucharest} \]

- Greedy search expands first the node that appears to be closest to the goal, according to \( h(n) \).
**A* search**

- **Idea:** avoid expanding paths that are already expensive

  evaluation function: \( f(n) = g(n) + h(n) \)

    - \( g(n) \) – cost so far to reach \( n \)
    - \( h(n) \) – estimated cost to goal from \( n \)
    - \( f(n) \) – estimated total cost of path through \( n \) to goal

- **A* search** uses an **admissible** heuristic, that is,

  \( h(n) \leq h^*(n) \) where \( h^*(n) \) is the **true** cost from \( n \).

  For example: \( h_{SLD}(n) \) never overestimates actual road distance.

- **Theorem:** A* search is optimal
Comparing uninformed search strategies

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-first cost</th>
<th>Uniform-depth-first</th>
<th>Depth-limited</th>
<th>Iterative deepening</th>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes if $l \geq d$</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- $b$ – max branching factor of the search tree
- $d$ – depth of the least-cost solution
- $m$ – max depth of the state-space (may be infinity)
- $l$ – depth cutoff
Comparing uninformed search strategies

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Greedy</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>(b^m) (at worst)</td>
<td>(b^m) (at worst)</td>
</tr>
<tr>
<td>Space</td>
<td>(b^m) (at worst)</td>
<td>(b^m) (at worst)</td>
</tr>
<tr>
<td>Optimal?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- \(b\) - max branching factor of the search tree
- \(d\) - depth of the least-cost solution
- \(m\) - max depth of the state-space (may be infinity)
- \(l\) - depth cutoff
Iterative improvement

- In many optimization problems, path is irrelevant; the goal state itself is the solution.

- In such cases, can use iterative improvement algorithms: keep a single “current” state, and try to improve it.
Hill climbing (or gradient ascent/descent)

- Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.

"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  local variables: current, a node
                  next, a node
  current ← MAKE-NODE(Initial-State[problem])
  loop do
    next ← a highest-valued successor of current
    if VALUE[next] < VALUE[current] then return current
    current ← next
  end
```
Consider how one might get a ball-bearing traveling along the curve to "probably end up" in the deepest minimum. The idea is to shake the box "about h hard" — then the ball is more likely to go from D to C than from C to D. So, on average, the ball should end up in C's valley.
Simulated annealing algorithm

- Idea: Escape local extrema by allowing “bad moves,” but gradually decrease their size and frequency.

```plaintext
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
             schedule, a mapping from time to “temperature”
    local variables: current, a node
                     next, a node
                     T, a “temperature” controlling the probability of downward steps
    current ← MAKE-NODE(Initial-State[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T=0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] - VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability $e^{ΔE/T}$
        Note: goal here is to maximize E.
```
Note on simulated annealing: limit cases

- **Boltzmann distribution:** accept “bad move” with $\Delta E < 0$ (goal is to maximize $E$) with probability $P(\Delta E) = \exp(\Delta E/T)$

- If $T$ is large: $\Delta E < 0$
  $\Delta E/T < 0$ and very small
  $\exp(\Delta E/T)$ close to 1
  accept bad move with high probability

- If $T$ is near 0: $\Delta E < 0$
  $\Delta E/T < 0$ and very large
  $\exp(\Delta E/T)$ close to 0
  accept bad move with low probability

Random walk

Deterministic
down-hill
Is search applicable to game playing?

- **Abstraction**: To describe a game we must capture every relevant aspect of the game. Such as:
  - Chess
  - Tic-tac-toe
  - ...

- **Accessible environments**: Such games are characterized by perfect information

- **Search**: Game-playing then consists of a search through possible game positions

- **Unpredictable opponent**: Introduces uncertainty thus game-playing must deal with contingency problems
Searching for the next move

- **Complexity:** many games have a huge search space
  - **Chess:** $b = 35, m=100 \Rightarrow \text{nodes} = 35^{100}$
    if each node takes about 1 ns to explore
    then each move will take about $10^{50}$ millennia
to calculate.

- **Resource (e.g., time, memory) limit:** optimal solution not feasible/possible, thus must approximate

1. **Pruning:** makes the search more efficient by discarding portions of the search tree that cannot improve quality result.

2. **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search.
The minimax algorithm

- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value  
  = best achievable payoff against best play

- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the tree by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use **minimax decision** to select the move with the max (of the min) utility value

- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.
minimax = maximum of the minimum
**α-β pruning: search cutoff**

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node.

- **α-β pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.

- Does it work? Yes, in roughly cuts the branching factor from $b$ to $\sqrt{b}$ resulting in double as far look-ahead than pure minimax.

- Important note: pruning does NOT affect the final result!
$\alpha$-$\beta$ pruning: example

MAX

MIN

$\geq 6$

6

6 12 8
\(\alpha - \beta\) pruning: example
$\alpha$-$\beta$ pruning: example

MAX

MIN

6 12 8

6 2

5

$\geq 6$

$\leq 2$

$\leq 5$
\( \alpha-\beta \) pruning: example

**MAX**

**MIN**

Selected move
Nondeterministic games: the element of chance

*expectimax* and *expectimin*, expected values over all possible outcomes
Nondeterministic games: the element of chance

Expectimax

MAX

Expectimin

MIN

4 = 0.5*3 + 0.5*5
Summary on games

Games are fun to work on! (and dangerous)

They illustrate several important points about AI

◊ perfection is unattainable ⇒ must approximate
◊ good idea to think about what to think about
◊ uncertainty constrains the assignment of values to states

Games are to AI as grand prix racing is to automobile design
Knowledge-Based Agent

- Agent that uses **prior** or **acquired** knowledge to achieve its goals
  - Can make more efficient decisions
  - Can make informed decisions
- Knowledge Base (KB): contains a set of **representations** of facts about the Agent’s environment
- Each representation is called a **sentence**
- Use some **knowledge representation language**, to TELL it what to know e.g., (temperature 72F)
- ASK agent to query what to do
- Agent can use inference to deduce new facts from TELLed facts
Generic knowledge-based agent

function KB-Agent(\texttt{percept}) returns an \texttt{action}
static: \texttt{KB}, a knowledge base
        \texttt{t}, a counter, initially 0, indicating time

\texttt{Tell}(\texttt{KB}, \texttt{Make-Percept-Sentence}(\texttt{percept}, \texttt{t}))
\texttt{action} \leftarrow \texttt{Ask}(\texttt{KB}, \texttt{Make-Action-Query}(\texttt{t}))
\texttt{Tell}(\texttt{KB}, \texttt{Make-Action-Sentence}(\texttt{action}, \texttt{t}))
\texttt{t} \leftarrow \texttt{t} + 1
\texttt{return} \texttt{action}

1. TELL KB what was perceived
   Uses a KRL to insert new sentences, representations of facts, into KB

2. ASK KB what to do.
   Uses logical reasoning to examine actions and select best.
Logic in general

Logics are formal languages for representing information such that conclusions can be drawn.

Syntax defines the sentences in the language.

Semantics define the “meaning” of sentences; i.e., define truth of a sentence in a world.

E.g., the language of arithmetic:

$x + 2 \geq y$ is a sentence; $x2 + y >$ is not a sentence.

$x + 2 \geq y$ is true iff the number $x + 2$ is no less than the number $y$.

$x + 2 \geq y$ is true in a world where $x = 7$, $y = 1$.

$x + 2 \geq y$ is false in a world where $x = 0$, $y = 6$. 
Types of logic

Logics are characterized by what they commit to as “primitives”


Epistemological commitment: what states of knowledge?

<table>
<thead>
<tr>
<th>Language</th>
<th>Ontological Commitment</th>
<th>Epistemological Commitment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional logic</td>
<td>facts</td>
<td>true/false/unknown</td>
</tr>
<tr>
<td>First-order logic</td>
<td>facts, objects, relations</td>
<td>true/false/unknown</td>
</tr>
<tr>
<td>Temporal logic</td>
<td>facts, objects, relations, times</td>
<td>true/false/unknown</td>
</tr>
<tr>
<td>Probability theory</td>
<td>facts</td>
<td>degree of belief 0...1</td>
</tr>
<tr>
<td>Fuzzy logic</td>
<td>degree of truth</td>
<td>degree of belief 0...1</td>
</tr>
</tbody>
</table>
Entailment

\[ KB \models \alpha \]

Knowledge base \( KB \) entails sentence \( \alpha \)

if and only if

\( \alpha \) is true in all worlds where \( KB \) is true

E.g., the KB containing "the Giants won" and "the Reds won" entails "Either the Giants won or the Reds won"
Inference

$KB \vdash_i \alpha$ = sentence $\alpha$ can be derived from $KB$ by procedure $i$

**Soundness:** $i$ is sound if
whenever $KB \vdash_i \alpha$, it is also true that $KB \models \alpha$

**Completeness:** $i$ is complete if
whenever $KB \models \alpha$, it is also true that $KB \vdash_i \alpha$

Preview: we will define a logic (first-order logic) which is expressive enough to say almost anything of interest, and for which there exists a sound and complete inference procedure.

That is, the procedure will answer any question whose answer follows from what is known by the $KB$. 
Validity and satisfiability

A sentence is **valid** if it is true in all models
 e.g., $A \lor \neg A$, $A \Rightarrow A$, $(A \land (A \Rightarrow B)) \Rightarrow$

Validity is connected to inference via the **Deduction Theorem**
 $KB \models \alpha$ if and only if $(KB \Rightarrow \alpha)$ is valid

A sentence is **satisfiable** if it is true in some model
 e.g., $A \lor B$, $C$

A sentence is **unsatisfiable** if it is true in no models
 e.g., $A \land \neg A$

Satisfiability is connected to inference via the following:
 $KB \models \alpha$ if and only if $(KB \land \neg \alpha)$ is unsatisfiable
 i.e., prove $\alpha$ by *reductio ad absurdum*
Propositional logic: semantics

Each model specifies true/false for each proposition symbol

E.g. \( A \quad B \quad C \)

\[ \text{True} \quad \text{True} \quad \text{False} \]

Rules for evaluating truth with respect to a model \( m \):

- \( \neg S \) is true iff \( S \) is false
- \( S_1 \land S_2 \) is true iff \( S_1 \) is true and \( S_2 \) is true
- \( S_1 \lor S_2 \) is true iff \( S_1 \) is true or \( S_2 \) is true
- \( S_1 \Rightarrow S_2 \) is true iff \( S_1 \) is false or \( S_2 \) is true
  i.e., is false iff \( S_1 \) is true and \( S_2 \) is false
- \( S_1 \Leftrightarrow S_2 \) is true iff \( S_1 \Rightarrow S_2 \) is true and \( S_2 \Rightarrow S_1 \) is true
Propositional inference: normal forms

Other approaches to inference use syntactic operations on sentences, often expressed in standardized forms.

**Conjunctive Normal Form (CNF—universal)**

- **Conjunction of disjunctions of literals**
- E.g., \((A \lor \neg B) \land (B \lor \neg C \lor \neg D)\)

**Disjunctive Normal Form (DNF—universal)**

- **Disjunction of conjunctions of literals**
- E.g., \((A \land B) \lor (A \land \neg C) \lor (A \land \neg D) \lor (\neg B \land \neg C) \lor (\neg B \land \neg D)\)

**Horn Form (restricted)**

- **Conjunction of Horn clauses** (clauses with \(\leq 1\) positive literal)
- E.g., \((A \lor \neg B) \land (B \lor \neg C \lor \neg D)\)
- Often written as set of implications:
  \(B \Rightarrow A\) and \((C \land D) \Rightarrow B\)
Proof methods

Proof methods divide into (roughly) two kinds:

Model checking
- truth table enumeration (sound and complete for propositional)
- heuristic search in model space (sound but incomplete)
  e.g., the GSAT algorithm (Ex. 6.15)

Application of inference rules
- Legitimate (sound) generation of new sentences from old
- Proof = a sequence of inference rule applications
  Can use inference rules as operators in a standard search alg.
◊ **Modus Ponens or Implication-Elimination:** (From an implication and the premise of the implication, you can infer the conclusion.)

\[
\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}
\]

◊ **And-Elimination:** (From a conjunction, you can infer any of the conjuncts.)

\[
\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i}
\]

◊ **And-Introduction:** (From a list of sentences, you can infer their conjunction.)

\[
\frac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}
\]

◊ **Or-Introduction:** (From a sentence, you can infer its disjunction with anything else at all.)

\[
\frac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n}
\]

◊ **Double-Negation Elimination:** (From a doubly negated sentence, you can infer a positive sentence.)

\[
\frac{\neg\neg\alpha}{\alpha}
\]

◊ **Unit Resolution:** (From a disjunction, if one of the disjuncts is false, then you can infer the other one is true.)

\[
\frac{\alpha \vee \beta, \quad \neg \beta}{\alpha}
\]

◊ **Resolution:** (This is the most difficult. Because \(\beta\) cannot be both true and false, one of the other disjuncts must be true in one of the premises. Or equivalently, implication is transitive.)

\[
\frac{\alpha \vee \beta, \quad \neg \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{or equivalently} \quad \frac{\neg \alpha \Rightarrow \beta, \quad \beta \Rightarrow \gamma}{\neg \alpha \Rightarrow \gamma}
\]
Limitations of Propositional Logic

1. It is too weak, i.e., has very limited expressiveness:
   - Each rule has to be represented for each situation:
     e.g., “don’t go forward if the wumpus is in front of you” takes 64 rules

2. It cannot keep track of changes:
   - If one needs to track changes, e.g., where the agent has been before then
     we need a timed-version of each rule. To track 100 steps we’ll then need
     6400 rules for the previous example.

It's hard to write and maintain such a huge rule-base
Inference becomes intractable
First-order logic (FOL)

- Ontological commitments:
  - **Objects**: wheel, door, body, engine, seat, car, passenger, driver
  - **Relations**: Inside(car, passenger), Beside(driver, passenger)
  - **Functions**: ColorOf(car)
  - **Properties**: Color(car), IsOpen(door), IsOn(engine)

- Functions are relations with single value for each object
Universal quantification (for all): $\forall$

$\forall \ <\text{variables}> \ <\text{sentence}>

- "Every one in the 561a class is smart":
  $\forall \ x \ \text{In}(561a, \ x) \Rightarrow \text{Smart}(x)$

- $\forall \ P \ \text{corresponds to the conjunction of instantiations of } P$
  $\text{In}(561a, \ \text{Manos}) \Rightarrow \text{Smart}(\text{Manos}) \land$
  $\text{In}(561a, \ \text{Dan}) \Rightarrow \text{Smart}(\text{Dan}) \land$
  $\ldots$
  $\text{In}(561a, \ \text{Clinton}) \Rightarrow \text{Smart}(\text{Mike})$

- $\Rightarrow$ is a natural connective to use with $\forall$
- **Common mistake:** to use $\land$ in conjunction with $\forall$
  e.g: $\forall \ x \ \text{In}(561a, \ x) \land \text{Smart}(x)$
  means "every one is in 561a and everyone is smart"
Existential quantification (there exists): \( \exists \)

\[ \exists \ <\text{variables}> <\text{sentence}> \]

- "Someone in the 561a class is smart":
  \[ \exists x \ \text{In}(561a, x) \land \text{Smart}(x) \]

- **P corresponds to the disjunction of instantiations of P**
  \[ \text{In}(561a, \text{Manos}) \land \text{Smart}(\text{Manos}) \lor \text{In}(561a, \text{Dan}) \land \text{Smart}(\text{Dan}) \lor \ldots \]
  \[ \text{In}(561a, \text{Clinton}) \land \text{Smart}(\text{Mike}) \]
  \[ \land \text{is a natural connective to use with } \exists \]

- **Common mistake**: to use \( \Rightarrow \) in conjunction with \( \exists \)
  e.g:
  \[ \exists x \ \text{In}(561a, x) \Rightarrow \text{Smart}(x) \]
  is true if there is anyone that is not in 561a!
  (remember, false \( \Rightarrow \) true is valid).
Properties of quantifiers

\[ \forall x \ \forall y \text{ is the same as } \forall y \ \forall x \ \text{ (why??)} \]

\[ \exists x \ \exists y \text{ is the same as } \exists y \ \exists x \ \text{ (why??)} \]

\[ \exists x \ \forall y \text{ is not the same as } \forall y \ \exists x \]

\[ \exists x \ \forall y \text{ Loves}(x, y) \]

"There is a person who loves everyone in the world"

\[ \forall y \ \exists x \text{ Loves}(x, y) \]

"Everyone in the world is loved by at least one person"

Quantifier duality: each can be expressed using the other

\[ \forall x \text{ Likes}(x, \text{IceCream}) \quad \neg \exists x \ \neg \text{Likes}(x, \text{IceCream}) \]

\[ \exists x \text{ Likes}(x, \text{Broccoli}) \quad \neg \forall x \ \neg \text{Likes}(x, \text{Broccoli}) \]
Example sentences

• Brothers are siblings

\[ \forall x, y \quad \text{Brother}(x, y) \Rightarrow \text{Sibling}(x, y) \]

• Sibling is transitive

\[ \forall x, y, z \quad \text{Sibling}(x, y) \land \text{Sibling}(y, z) \Rightarrow \text{Sibling}(x, z) \]

• One’s mother is one’s sibling’s mother

\[ \forall m, c \quad \text{Mother}(m, c) \land \text{Sibling}(c, d) \Rightarrow \text{Mother}(m, d) \]

• A first cousin is a child of a parent’s sibling

\[ \forall c, d \quad \text{FirstCousin}(c, d) \iff \\
\exists p, ps \quad \text{Parent}(p, d) \land \text{Sibling}(p, ps) \land \text{Parent}(ps, c) \]
Higher-order logic?

- First-order logic allows us to quantify over objects (= the first-order entities that exist in the world).

- Higher-order logic also allows quantification over relations and functions.
  e.g., “two objects are equal iff all properties applied to them are equivalent”:

\[ \forall x, y \ (x = y) \iff (\forall p, p(x) \iff p(y)) \]

- Higher-order logics are more expressive than first-order; however, so far we have little understanding on how to effectively reason with sentences in higher-order logic.
Using the FOL Knowledge Base

Suppose a wumpus-world agent is using an FOL KB and perceives a smell and a breeze (but no glitter) at \( t = 5 \):

\[
\text{Tell}(KB, \text{Percept}([\text{Smell}, \text{Breeze}, \text{None}], 5)) \\
\text{Ask}(KB, \exists a \ \text{Action}(a, 5))
\]

I.e., does the KB entail any particular actions at \( t = 5 \)?

Answer: Yes, \( \{a/\text{Shoot}\} \leftarrow \text{substitution} \) (binding list)

Given a sentence \( S \) and a substitution \( \sigma \), \( S\sigma \) denotes the result of plugging \( \sigma \) into \( S \); e.g.,

\[
S = \text{Smarter}(x, y) \\
\sigma = \{x/\text{Hillary}, y/\text{Bill}\} \\
S\sigma = \text{Smarter}(\text{Hillary}, \text{Bill})
\]

\text{Ask}(KB, S) \text{ returns some/all } \sigma \text{ such that } KB \models S\sigma
Wumpus world, FOL Knowledge Base

“Perception”
\[ \forall b, g, t \ Percept([Smell, b, g], t) \Rightarrow Smelt(t) \]
\[ \forall s, b, t \ Percept([s, b, Glitter], t) \Rightarrow AtGold(t) \]

Reflex: \[ \forall t \ AtGold(t) \Rightarrow Action(Grab, t) \]

Reflex with internal state: do we have the gold already?
\[ \forall t \ AtGold(t) \land \neg Holding(Gold, t) \Rightarrow Action(Grab, t) \]

\textit{Holding}(Gold, t) cannot be observed
\[ \Rightarrow \text{keeping track of change is essential} \]
Deducing hidden properties

Properties of locations:
\[ \forall l, t \ (At(Agent, l, t) \land Smelt(t) \Rightarrow Smelly(l)) \]
\[ \forall l, t \ (At(Agent, l, t) \land Breeze(t) \Rightarrow Breezy(l)) \]

Squares are breezy near a pit:

Diagnostic rule—infer cause from effect
\[ \forall y \ Breezy(y) \Rightarrow \exists x \ Pit(x) \land Adjacent(x, y) \]

Causal rule—infer effect from cause
\[ \forall x, y \ Pit(x) \land Adjacent(x, y) \Rightarrow Breezy(y) \]

Neither of these is complete—e.g., the causal rule doesn’t say whether squares far away from pits can be breezy

Definition for the Breezy predicate:
\[ \forall y \ Breezy(y) \iff [\exists x \ Pit(x) \land Adjacent(x, y)] \]
Situation calculus

Facts hold in situations, rather than eternally
E.g., $\textit{Holding}(\text{Gold}, \text{Now})$ rather than just $\textit{Holding}(\text{Gold})$

\textbf{Situation calculus} is one way to represent change in FOL:
   Adds a situation argument to each non-eternal predicate
E.g., $\text{Now}$ in $\textit{Holding}(\text{Gold}, \text{Now})$ denotes a situation

Situations are connected by the $\text{Result}$ function
$\text{Result}(a, s)$ is the situation that results from doing $a$ is $s$
Describing actions

“Effect” axiom—describe changes due to action
\[ \forall s \ \text{AtGold}(s) \Rightarrow \text{Holding}(Gold, \text{Result}(\text{Grab}, s)) \]

“Frame” axiom—describe non-changes due to action
\[ \forall s \ \text{HaveArrow}(s) \Rightarrow \text{HaveArrow}(\text{Result}(\text{Grab}, s)) \]

Frame problem: find an elegant way to handle non-change
(a) representation—avoid frame axioms
(b) inference—avoid repeated “copy-overs” to keep track of state

Qualification problem: true descriptions of real actions require endless caveats—what if gold is slippery or nailed down or . . .

Ramification problem: real actions have many secondary consequences—what about the dust on the gold, wear and tear on gloves, . . .
Describing actions (cont’d)

**Successor-state axioms** solve the representational frame problem

Each axiom is “about” a **predicate** (not an action per se):

\[ P \text{ true afterwards } \Leftrightarrow \ [\text{an action made } P \text{ true} \]

\[ \lor P \text{ true already and no action made } P \text{ false} \]

For holding the gold:

\[ \forall a, s \ Holding(Gold, Result(a, s)) \Leftrightarrow \]

\[ [(a = Grab \land AtGold(s)) \]

\[ \lor (Holding(Gold, s) \land a \neq Release)] \]
Planning

Initial condition in KB:
\[ At(Agent, [1, 1], S_0) \]
\[ At(Gold, [1, 2], S_0) \]

Query: \[ \text{Ask}(KB, \exists s \; \text{Holding}(Gold, s)) \]

i.e., in what situation will I be holding the gold?

Answer: \[ \{s/\text{Result(Grab, Result(Forward, S_0))}\} \]

i.e., go forward and then grab the gold

This assumes that the agent is interested in plans starting at \( S_0 \) and that \( S_0 \) is the only situation described in the KB
Generating action sequences

Represent plans as action sequences \([a_1, a_2, \ldots, a_n]\)

\(\text{PlanResult}(p, s)\) is the result of executing \(p\) in \(s\)

Then the query \(\text{Ask}(KB, \exists p \ \text{Holding}(Gold, \text{PlanResult}(p, S_0)))\) has the solution \([p/ [\text{Forward, Grab}]]\)

Definition of \(\text{PlanResult}\) in terms of \(\text{Result}\):
\[
\forall s \ \text{PlanResult}([], s) = s
\]
\[
\forall a, p, s \ \text{PlanResult}([a|p], s) = \text{PlanResult}(p, \text{Result}(a, s))
\]

Planning systems are special-purpose reasoners designed to do this type of inference more efficiently than a general-purpose reasoner.
Summary on FOL

First-order logic:
- objects and relations are semantic primitives
- syntax: constants, functions, predicates, equality, quantifiers

Increased expressive power: sufficient to define wumpus world

Situation calculus:
- conventions for describing actions and change in FOL
- can formulate planning as inference on a situation calculus KB
Knowledge Engineer

- Populates KB with facts and relations

- Must study and understand domain to pick important objects and relationships

- **Main steps:**
  
  Decide what to talk about
  Decide on vocabulary of predicates, functions & constants
  Encode general knowledge about domain
  Encode description of specific problem instance
  Pose queries to inference procedure and get answers
## Knowledge engineering vs. programming

<table>
<thead>
<tr>
<th>Knowledge Engineering</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Choosing a logic</td>
<td>Choosing programming language</td>
</tr>
<tr>
<td>2. Building knowledge base</td>
<td>Writing program</td>
</tr>
<tr>
<td>3. Implementing proof theory</td>
<td>Choosing/writing compiler</td>
</tr>
<tr>
<td>4. Inferring new facts</td>
<td>Running program</td>
</tr>
</tbody>
</table>

Why knowledge engineering rather than programming?

**Less work:** just specify objects and relationships known to be true, but leave it to the inference engine to figure out how to solve a problem using the known facts.
Towards a general ontology

• Develop good representations for:

- categories
- measures
- composite objects
- time, space and change
- events and processes
- physical objects
- substances
- mental objects and beliefs
- …
Inference in First-Order Logic

- Proofs – extend propositional logic inference to deal with quantifiers

- Unification
- Generalized modus ponens
- Forward and backward chaining – inference rules and reasoning program
- Completeness – Gödel’s theorem: for FOL, any sentence entailed by another set of sentences can be proved from that set
- Resolution – inference procedure that is complete for any set of sentences
- Logic programming
Proofs

The three new inference rules for FOL (compared to propositional logic) are:

• **Universal Elimination (UE):**
  for any sentence $\alpha$, variable $x$ and ground term $\tau$,
  \[ \forall x \alpha \quad \underline{\alpha \{x/\tau\}} \]
e.g., from $\forall x \text{Likes}(x, \text{Candy})$ and $\{x/\text{Joe}\}$ we can infer $\text{Likes}(\text{Joe, Candy})$

• **Existential Elimination (EE):**
  for any sentence $\alpha$, variable $x$ and constant symbol $k$ not in KB,
  \[ \exists x \alpha \quad \underline{\alpha \{x/k\}} \]
e.g., from $\exists x \text{Kill}(x, \text{Victim})$ we can infer $\text{Kill}($\text{Murderer, Victim}$)$, if Murderer new symbol

• **Existential Introduction (EI):**
  for any sentence $\alpha$, variable $x$ not in $\alpha$ and ground term $g$ in $\alpha$,
  \[ \underline{\alpha} \quad \exists x \alpha \{g/x\} \]
e.g., from Likes(\text{Joe, Candy}) we can infer $\exists x \text{Likes}(x, \text{Candy})$
Generalized Modus Ponens (GMP)

\[
p_1', \ p_2', \ \ldots, \ p_n', \ (p_1 \land p_2 \land \ldots \land p_n \Rightarrow q) \quad \frac{\text{where } p_i'\sigma = p_i\sigma \text{ for all } i}{q\sigma}
\]

E.g. \( p_1' = \text{Faster(Bob,Pat)} \)
\( p_2' = \text{Faster(Pat,Steve)} \)
\( p_1 \land p_2 \Rightarrow q = \text{Faster}(x, y) \land \text{Faster}(y, z) \Rightarrow \text{Faster}(x, z) \)
\( \sigma = \{x/\text{Bob}, y/\text{Pat}, z/\text{Steve}\} \)
\( q\sigma = \text{Faster}(\text{Bob, Steve}) \)

GMP used with KB of definite clauses (exactly one positive literal):
either a single atomic sentence or
(conjunction of atomic sentences) \( \Rightarrow \) (atomic sentence)
All variables assumed universally quantified
When a new fact $p$ is added to the KB
for each rule such that $p$ unifies with a premise
if the other premises are known
then add the conclusion to the KB and continue chaining

Forward chaining is data-driven
  e.g., inferring properties and categories from percepts
Backward chaining

When a query $q$ is asked

if a matching fact $q'$ is known, return the unifier
for each rule whose consequent $q'$ matches $q$
attempt to prove each premise of the rule by backward chaining

(Some added complications in keeping track of the unifiers)

(More complications help to avoid infinite loops)

Two versions: find any solution, find all solutions

Backward chaining is the basis for logic programming, e.g., Prolog
Resolution

Entailment in first-order logic is only **semidecidable**:
- can find a proof of $\alpha$ if $KB \models \alpha$
- cannot always prove that $KB \not\models \alpha$

Cf. Halting Problem: proof procedure may be about to terminate with success or failure, or may go on for ever

Resolution is a **refutation** procedure:
- to prove $KB \models \alpha$, show that $KB \land \neg \alpha$ is unsatisfiable

Resolution uses $KB$, $\neg \alpha$ in CNF (conjunction of clauses)

Resolution inference rule combines two clauses to make a new one:

\[
\begin{array}{c}
C_1 \rightarrow \\
C_2 \\
\hline
C
\end{array}
\]

Inference continues until an **empty clause** is derived (contradiction)
Resolution inference rule

Basic propositional version:

\[
\frac{\alpha \lor \beta, \; \neg \beta \lor \gamma}{\alpha \lor \gamma}
\]

or equivalently

\[
\frac{\neg \alpha \Rightarrow \beta, \; \beta \Rightarrow \gamma}{\neg \alpha \Rightarrow \gamma}
\]

Full first-order version:

\[
\begin{align*}
& p_1 \lor \ldots \lor p_j \ldots \lor p_m, \\
& q_1 \lor \ldots \lor q_k \ldots \lor q_n
\end{align*}
\]

\[
\frac{(p_1 \lor \ldots \lor p_{j-1} \lor p_{j+1} \ldots \lor p_m \lor q_1 \ldots \lor q_{k-1} \lor q_{k+1} \ldots \lor q_n) \sigma}{\neg p_j \sigma = \neg q_k \sigma}
\]

where \( p_j \sigma = \neg q_k \sigma \)

For example,

\[
\begin{align*}
& \neg Rich(x) \lor Unhappy(x) \\
& Rich(Me) \\
\end{align*}
\]

\[
\begin{align*}
& \frac{Unhappy(Me)}{Unhappy(Me)}
\end{align*}
\]

with \( \sigma = \{x/Me\} \)
Resolution proof

To prove \( \alpha \):
- negate it
- convert to CNF
- add to CNF KB
- infer contradiction

E.g., to prove \( \text{Rich}(me) \), add \( \neg \text{Rich}(me) \) to the CNF KB

\[
\neg \text{PhD}(x) \lor \text{HighlyQualified}(x) \\
\text{PhD}(x) \lor \text{EarlyEarnings}(x) \\
\neg \text{HighlyQualified}(x) \lor \text{Rich}(x) \\
\neg \text{EarlyEarnings}(x) \lor \text{Rich}(x)
\]
Logical reasoning systems

• Theorem provers and logic programming languages

• Production systems

• Frame systems and semantic networks

• Description logic systems
Logical reasoning systems

• **Theorem provers and logic programming languages** – Provers: use resolution to prove sentences in full FOL. Languages: use backward chaining on restricted set of FOL constructs.

• **Production systems** – based on implications, with consequents interpreted as action (e.g., insertion & deletion in KB). Based on forward chaining + conflict resolution if several possible actions.

• **Frame systems and semantic networks** – objects as nodes in a graph, nodes organized as taxonomy, links represent binary relations.

• **Description logic systems** – evolved from semantic nets. Reason with object classes & relations among them.
Membership functions: S-function

- The S-function can be used to define fuzzy sets
- \( S(x, a, b, c) = \)
  - 0 \quad \text{for } x \leq a
  - 2(x-a/c-a)^2 \quad \text{for } a \leq x \leq b
  - 1 - 2(x-c/c-a)^2 \quad \text{for } b \leq x \leq c
  - 1 \quad \text{for } x \geq c

![Graph of S-function](image)
Membership functions: \( \Pi \)-Function

- \( \Pi(x, a, b) = \)
  - \( S(x, b-a, b-a/2, b) \) for \( x \leq b \)
  - \( 1 - S(x, b, b+a/2, a+b) \) for \( x \geq b \)

E.g., *close* (to \( a \))
Linguistic Hedges

- Modifying the meaning of a fuzzy set using hedges such as *very*, *more or less*, *slightly*, *etc.*

- Very $F = F^2$
- More or less $F = F^{1/2}$
- etc.

![Diagram of height vs. membership function showing 'tall', 'more or less tall', and 'very tall' curves.](image)
Fuzzy set operators

- Equality
  \( A = B \)
  \[ \mu_A(x) = \mu_B(x) \quad \text{for all } x \in X \]

- Complement
  \( A' \)
  \[ \mu_{A'}(x) = 1 - \mu_A(x) \quad \text{for all } x \in X \]

- Containment
  \( A \subseteq B \)
  \[ \mu_A(x) \leq \mu_B(x) \quad \text{for all } x \in X \]

- Union
  \( A \cup B \)
  \[ \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x)) \quad \text{for all } x \in X \]

- Intersection
  \( A \cap B \)
  \[ \mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x)) \quad \text{for all } x \in X \]
Fuzzy inference overview

1. fuzzify inputs
   - poor
   - rancid

   If service is poor or food is rancid then tip = cheap

2. apply fuzzy operation (or = max)
   - cheap

   If service is good then tip = average

   Rule 2 has no dependency on input 2

3. apply implication method (min)
   - average

   If service is excellent or food is delicious then tip = generous

4. apply aggregation method (max)

   Tip = 16.7%
   Result of defuzzification (centroid)
What we have so far

• Can TELL KB about new percepts about the world

• KB maintains model of the current world state

• Can ASK KB about any fact that can be inferred from KB

How can we use these components to build a planning agent, i.e., an agent that constructs plans that can achieve its goals, and that then executes these plans?
Search vs. planning

Consider the task *get milk, bananas, and a cordless drill*

Standard search algorithms seem to fail miserably:

After-the-fact heuristic/goal test inadequate
Types of planners

- Situation space planner: search through possible situations

- Progression planner: start with initial state, apply operators until goal is reached
  Problem: high branching factor!

- Regression planner: start from goal state and apply operators until start state reached
  Why desirable? usually many more operators are applicable to initial state than to goal state.
  Difficulty: when want to achieve a conjunction of goals

Initial STRIPS algorithm: situation-space regression planner
A Simple Planning Agent

function SIMPLE-PLANNING-AGENT(percept) returns an action

static: KB, a knowledge base (includes action descriptions)
        p, a plan (initially, NoPlan)
        t, a time counter (initially 0)

local variables: G, a goal
                current, a current state description

TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
current ← STATE-DESCRIPTION(KB, t)

if p = NoPlan then
    G ← ASK(KB, MAKE-GOAL-QUERY(t))
    p ← IDEAL-PLANNER(current, G, KB)
if p = NoPlan or p is empty then
    action ← NoOp
else
    action ← FIRST(p)
    p ← REST(p)
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t ← t+1
return action
STRIPS operators

Tidily arranged actions descriptions, restricted language

**ACTION:** \textit{Buy}(x)

**PRECONDITION:** \textit{At}(p), \textit{Sells}(p, x)

**EFFECT:** \textit{Have}(x)

[Note: this abstracts away many important details!]

Restricted language $\Rightarrow$ efficient algorithm

Precondition: conjunction of positive literals

Effect: conjunction of literals

Graphical notation:

\[
\begin{array}{c}
\text{At}(p) \quad \text{Sells}(p, x) \\
\text{Buy}(x) \\
\text{Have}(x)
\end{array}
\]
A plan is complete iff every precondition is achieved.

A precondition is achieved iff it is the effect of an earlier step and no possibly intervening step undoes it.
Plan

We formally define a plan as a data structure consisting of:

- Set of plan steps (each is an operator for the problem)

- Set of step ordering constraints

  e.g., $A \sqsupset B$ means “$A$ before $B$”

- Set of variable binding constraints

  e.g., $v = x$ where $v$ variable and $x$ constant or other variable

- Set of causal links

  e.g., $A \xrightarrow{c} B$ means “$A$ achieves $c$ for $B$”
POP algorithm sketch

function POP(initial, goal, operators) returns plan

    plan ← MAKE-MINIMAL-PLAN(initial, goal)
    loop do
        if SOLUTION?(plan) then return plan
        S_{need}, c ← SELECT-SUBGOAL(plan)
        CHOOSE-OPERATOR(plan, operators, S_{need}, c)
        RESOLVE-THREATS(plan)
    end

function SELECT-SUBGOAL(plan) returns S_{need}, c

    pick a plan step S_{need} from STEPS(plan)
    with a precondition c that has not been achieved
    return S_{need}, c
POP algorithm (cont.)

```plaintext
procedure Choose-Operator(\textit{plan}, \textit{operators}, S_{\text{need}}, c)
    
    choose a step $S_{\text{add}}$ from \textit{operators} or STEPS(\textit{plan}) that has \textit{c} as an effect 
    if there is no such step then fail 
    add the causal link $S_{\text{add}} \rightarrow c \rightarrow S_{\text{need}}$ to LINKS(\textit{plan}) 
    add the ordering constraint $S_{\text{add}} < S_{\text{need}}$ to ORDERINGS(\textit{plan}) 
    if $S_{\text{add}}$ is a newly added step from \textit{operators} then 
        add $S_{\text{add}}$ to STEPS(\textit{plan}) 
        add $\text{Start} < S_{\text{add}} < \text{Finish}$ to ORDERINGS(\textit{plan})

procedure Resolve-Threats(\textit{plan})
    for each $S_{\text{threat}}$ that threatens a link $S_i \rightarrow c \rightarrow S_j$ in LINKS(\textit{plan}) do 
        choose either 
        Demotion: Add $S_{\text{threat}} < S_i$ to ORDERINGS(\textit{plan}) 
        Promotion: Add $S_j < S_{\text{threat}}$ to ORDERINGS(\textit{plan}) 
        if not CONSISTENT(\textit{plan}) then fail 
    end

POP is sound, complete, and \underline{systematic} (no repetition)

Extensions for disjunction, universals, negation, conditionals
```
Some problems remain...

- Vision
- Audition / speech processing
- Natural language processing
- Touch, smell, balance and other senses
- Motor control

They are extensively studied in other courses.
Computer Perception

- **Perception**: provides an agent information about its environment. Generates feedback. Usually proceeds in the following steps.

  1. **Sensors**: hardware that provides raw measurements of properties of the environment
     1. Ultrasonic Sensor/Sonar: provides distance data
     2. Light detectors: provide data about intensity of light
     3. Camera: generates a picture of the environment
  2. **Signal processing**: to process the raw sensor data in order to extract certain features, e.g., color, shape, distance, velocity, etc.
  3. **Object recognition**: Combines features to form a model of an object
  4. And so on to higher abstraction levels
Perception for what?

- **Interaction** with the environment, e.g., manipulation, navigation
- **Process control**, e.g., temperature control
- **Quality control**, e.g., electronics inspection, mechanical parts
- **Diagnosis**, e.g., diabetes
- **Restoration**, of e.g., buildings
- **Modeling**, of e.g., parts, buildings, etc.
- **Surveillance**, banks, parking lots, etc.
- ...
- And much, much more
Image analysis/Computer vision

1. Grab an image of the object (digitize analog signal)

2. Process the image (looking for certain features)
   1. Edge detection
   2. Region segmentation
   3. Color analysis
   4. Etc.

3. Measure properties of features or collection of features (e.g., length, angle, area, etc.)

4. Use some model for detection, classification etc.
Visual Attention
Pedestrian recognition

- C. Papageorgiou & T. Poggio, MIT
More robot examples

Rhex, U. Michigan
A McCulloch-Pitts neuron operates on a discrete time-scale, \( t = 0,1,2,3, \ldots \) with time tick equal to one refractory period.

At each time step, an input or output is

\[ \text{on or off} \quad 1 \text{ or } 0, \text{ respectively.} \]

Each connection or synapse from the output of one neuron to the input of another, has an attached weight.
Leaky Integrator Neuron

- The simplest "realistic" neuron model is a continuous time model based on using the firing rate (e.g., the number of spikes traversing the axon in the most recent 20 msec.) as a continuously varying measure of the cell's activity.

- The state of the neuron is described by a single variable, the membrane potential.

- The firing rate is approximated by a sigmoid, function of membrane potential.

\[
M(t) = \sigma(m(t))
\]
Leaky Integrator Model

\[ \tau \frac{d}{dt} m(t) = -m(t) + h \]

has solution

\[ m(t) = e^{-t/\tau} m(0) + (1 - e^{-t/\tau})h \]

\[ \rightarrow h \text{ for time constant } \tau > 0. \]

- We now add synaptic inputs to get the

**Leaky Integrator Model:**

\[ \tau \frac{d}{dt} m(t) = -m(t) + \sum_i w_i X_i(t) + h \]

where \( X_i(t) \) is the firing rate at the \( i^{th} \) input.

- Excitatory input \( (w_i > 0) \) will increase \( m(t) \)
- Inhibitory input \( (w_i < 0) \) will have the opposite effect.
Hopfield Networks

- A Hopfield net (Hopfield 1982) is a net of such units subject to the asynchronous rule for updating one neuron at a time:

  "Pick a unit i at random.
  If \( \sum w_{ij} s_j \geq \theta_i \), turn it on.
  Otherwise turn it off."

- Moreover, Hopfield assumes symmetric weights:
  \( w_{ij} = w_{ji} \)
“Energy” of a Neural Network

• Hopfield defined the “energy”:

\[ E = - \frac{1}{2} \sum_{ij} s_i s_j w_{ij} + \sum_i s_i \theta_i \]

• If we pick unit i and the firing rule (previous slide) does not change its \( s_i \), it will not change \( E \).
Self-Organizing Feature Maps

- The neural sheet is represented in a discretized form by a (usually) 2-D lattice $A$ of formal neurons.

- The input pattern is a vector $x$ from some pattern space $V$. Input vectors are normalized to unit length.

- The responsiveness of a neuron at a site $r$ in $A$ is measured by $x.w_r = \sum_i x_i w_{ri}$

- The "image" of an external event is regarded as the unit with the maximal response to it.
Example: face recognition

- Here using the 2-stage approach:
Associative Memories

- [http://www.shef.ac.uk/psychology/gurney/notes/5615.html](http://www.shef.ac.uk/psychology/gurney/notes/5615.html)

- Idea: store:

So that we can recover it if presented with corrupted data such as:

Original 'T'

half of image corrupted by noise

20% corrupted by noise (whole image)